F.) Specification of the Fleas symmetric ciphers

Overview

The Fleas, flight, flightx symmetric ciphers have been newly developed for use in the cryptographic software package "Academic Signature". (The chimera cipher is different and is treated in a separate document).They are based on different principles than traditional ciphers and do not use the classical permutation substitution techniques like e.g. "S-boxes" or the like. The new ciphers employ naturally arising entropy increases by self referential development of block/key concatenations as realized in the Fleas pseudorandom functions. They are to be used with substantially increased block size compared to traditional ciphers. The newly developed ciphers serve the purpose of allowing maximum independence of US/NSA influenced standards as well as maximum self containedness of the package "Academic Signature".

The Fleas symmetric ciphers are accessible directly in Academic Signature from the menu entry "Accessories → Hardened Symmetric Crypto". Additionally they are used in elliptic curve ciphers as basic symmetric encryption method using a key which is enciphered asymmetrically. Furthermore they are used internally to protect the secrets in Academic Signature like private keys, public keys and PRNG recovery files.

All the routines employed for symmetric enciphering can be found in the C++ module helpersxx.cpp. Stretching, as implemented in Academic Signature, involves elliptic curve calculations. Thus the implementation of the procedure "kyprep" which achieves stretching (and salting) resides in the module "elliptic.cpp".

If used directly for symmetric ciphers, special precautions are taken to harden the ciphers against dictionary attacks possibly launched by high budget organizations. This is done by employing a new variant of stretching and by employing salting in the derivation of a pseudorandom key from a human memorizable primary keyword. The size of salts and the complexity of stretching calculations exceed the size and amount used in conventional cryptographic applications. In the direct usage of symmetric ciphers, cipher integrity is assured by employing an HMAC.

In contrast with directly accessed symmetrical enciphering, in ecc-ciphers salting and stretching is not necessary. The symmetric key is derived from a random number in that case. Likewise in internal use of symmetric ciphers, additional stretching and salting of the key is unnecessary since the key already is the product of a process hat involved salting and stretching and had been derived from the human memorizable login passphrase used for entry into Academic Signature.

For all uses the symmetric ciphers may be used in the cipher block chaining mode and in counter mode.

a) Salting and Stretching in direct Use of Symmetric Ciphers

The procedure "kyprep", which performs stretching and salting is located in the module "elliptic.cpp". The C++ code is shown below:

```
/*****************************************************/
bool kyprep(longnumber *pky, wxString *keystr, longnumber *psalt, ellipse *pse,  int stretch,int
outlen, bool legacy )
//produce outlen byte key from ky
{
    longnumber tmp;
    e_p point;
    char *tmpc;
    int i;
    bool goodp;

    // if default return trivial
    if(legacy) pky->stortext_old(keystr);
    else pky->stortext(keystr);
    if((*keystr== _("default"))||( (psalt==NULL)&&(stretch==0) ))
    {
        return true; //cut short if default
    }
    pky->setsize(true);
    pky->shrinktofit(1);
    if(psalt != NULL) pky->appendnum(psalt); //mingle with salt prior to time consuming step
    pky->setsize(true);
    pky->shrinktofit(1);
    if((outlen>2000)||(outlen<5)){throwout(_("Fatal error!\nkeylength > 2000 bytes or <5 byte is
crazy\n aborting!")); return false;}
    tmp.resizelonu((unsigned long)(outlen+2),0);
    //expand key into outlen byte with oneway fun from wxstring
    tmpc=develop_o_malloc((int) pky->size, (char *)(pky->ad), 2,5,outlen,2);
    memcpy(tmp.ad,tmpc,outlen);
    free(tmpc);
    tmp.setsize(true);
    if(pse!=NULL)
    {
//initialize Progress bar
       wxProgressDialog pbar(_("stretching"),_("elliptic curve stretching in progress\nplease
wait!"),stretch);
       for(i=0;i<stretch;i++) //time consuming operations
       {
           goodp=pbar.Update(i+1u);
         tmp.lonumodulo_qqq_e(&(pse->q)); //modulo group order q
           //elliptic operations
           point.copy_ep(&(pse->d0));
           point.mult_p_qj(&tmp, pse);
            //put y-coordinate into tmp
            tmp.copynum(&(point.y));
            tmp.shrinktofit(1);
            tmp.appendnum(&(point.x));
            tmp.setsize(true);
            tmp.shrinktofit(1);
       }
    }
    if(psalt != NULL) tmp.randomize_o(2,psalt,(unsigned long)outlen,0,0);//mingle with salt again
    else tmp.randomize_d(2,0);
    tmp.resizelonu((unsigned long)(outlen+2),0);
    tmp.setsize(true);
    //expand key into outlen byte with oneway fun from wxstring
    tmpc=develop_o_malloc((int) tmp.size, (char *) tmp.ad, 2,5,outlen,2);
    memcpy(tmp.ad,tmpc,outlen);
    free(tmpc);
    tmp.setsize(true);
    pky->copynum(&tmp);
    //if(
    return true;
}
/**********************************************************/
```

Comments on the code:

Upon encountering the escape passphrase "default", salting and stretching would be bypassed.

The salt (32 byte) is appended to the raw key.

The raw key/salt block is co-randomized to a 512 byte block using the pseudorandom function "develop_o".

Then the following loop is executed:

Interpret the block as a longnumber(low byte at low address) and modulo reduce the block with respect to group order of the elliptic curve used for stretching.
Multiply the generator of the elliptic curve with this number.
Concatenate y- and x-coordinate of the resulting point to a new longnumber.
Repeat loop "stretch factor" times.

Upon leaving the loop, co-randomize (and expand) the resulting longnumber with the salt to obtain a 512 byte block.

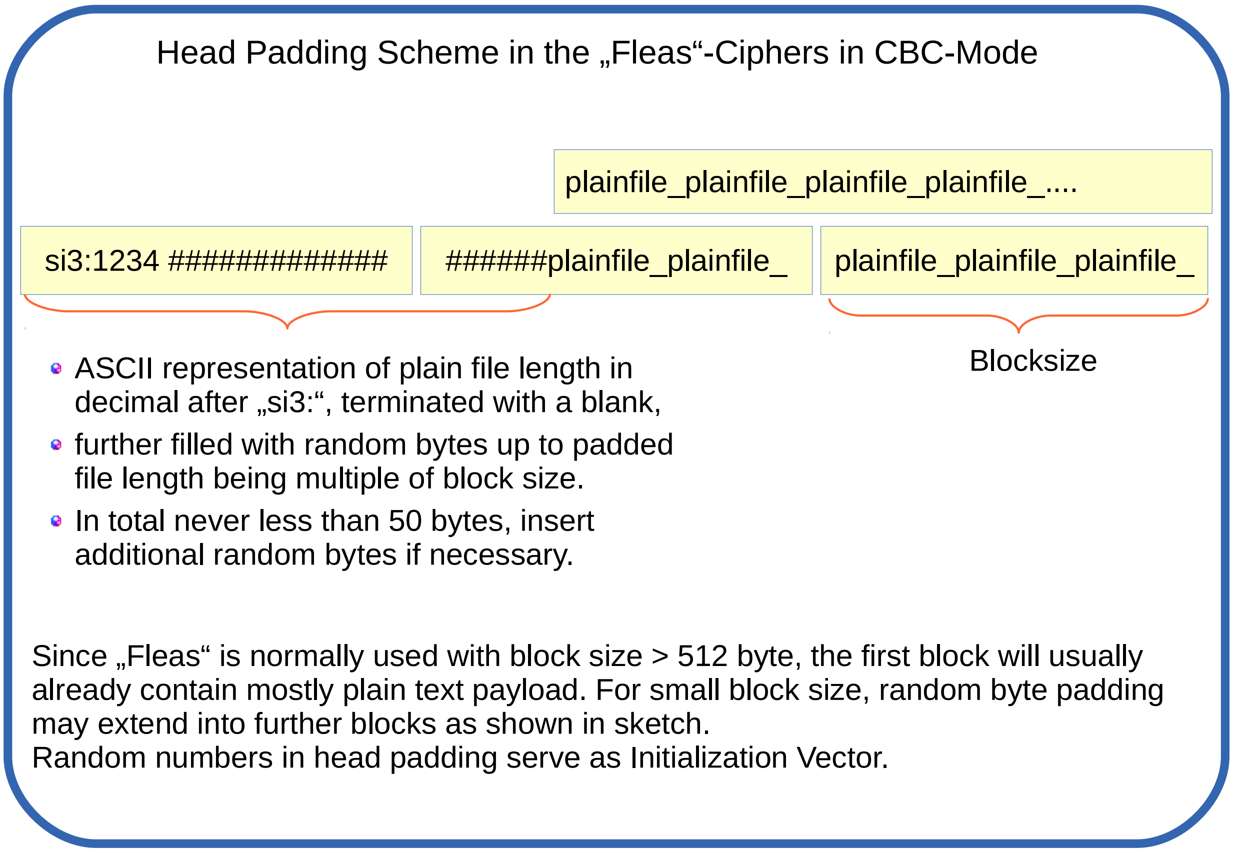
Apply pseudorandom function "develop_o" once more to receive a processed key of 512 byte.

Use this key for whatever purpose in the calling procedure.

In Academic Signature the elliptic curve used for stretching is a 253 bit domain without special form that would allow for speed enhancements. The domain is hard coded in the program Academic Signature.

b) Using Fleas-ciphers in Cipher Block Chaining Mode

For usage in CBC mode, padding to a multiple of the block size is necessary. Furthermore a unique initialization vector is needed to enforce individually different ciphers for multiple encipherings of the same plaintext. The following diagram shows how this is achieved.


Head Padding Scheme in the „Fleas“-Ciphers in CBC-Mode
plainfile_plainfile_plainfile_plainfile_....
si3:1234 ##############
######plainfile_plainfile_
plainfile_plainfile_plainfile_
Blocksize

- ASCII representation of plain file length in decimal after „si3:“, terminated with a blank,
- further filled with random bytes up to padded file length being multiple of block size.
- In total never less than 50 bytes, insert additional random bytes if necessary.

Since „Fleas“ is normally used with block size > 512 byte, the first block will usually already contain mostly plain text payload. For small block size, random byte padding may extend into further blocks as shown in sketch.
Random numbers in head padding serve as Initialization Vector.

After padding, the file is enciphered using one of the presently accessible modes:

Fleas_1_8, Fleas_3, Fleas_o2, Fleas_o5, Fleas_l, Fleas_ls, Fleas_l3, Fleas_lc and Fleas_ld. Apart from the variants Fleas_1_8 and Fleas_3, which generally use a block size of 130 bytes, all other variants are usually used with a bit size beyond 512 byte. Note that the algorithms can all be used with arbitrary block size that has to be even, however, when using CBC mode and the Feistel network. I consider the algorithms secure if used with a block size above about 32 byte. Fleas_l is a two path version and the fastest algorithm containing the least security overkill. Fleas_o5 is a five paths version and employs a security overkill bordering to the unreasonable, suitable for the most paranoid of users. It is used internally e.g. to encipher private key information for transfer to other systems.

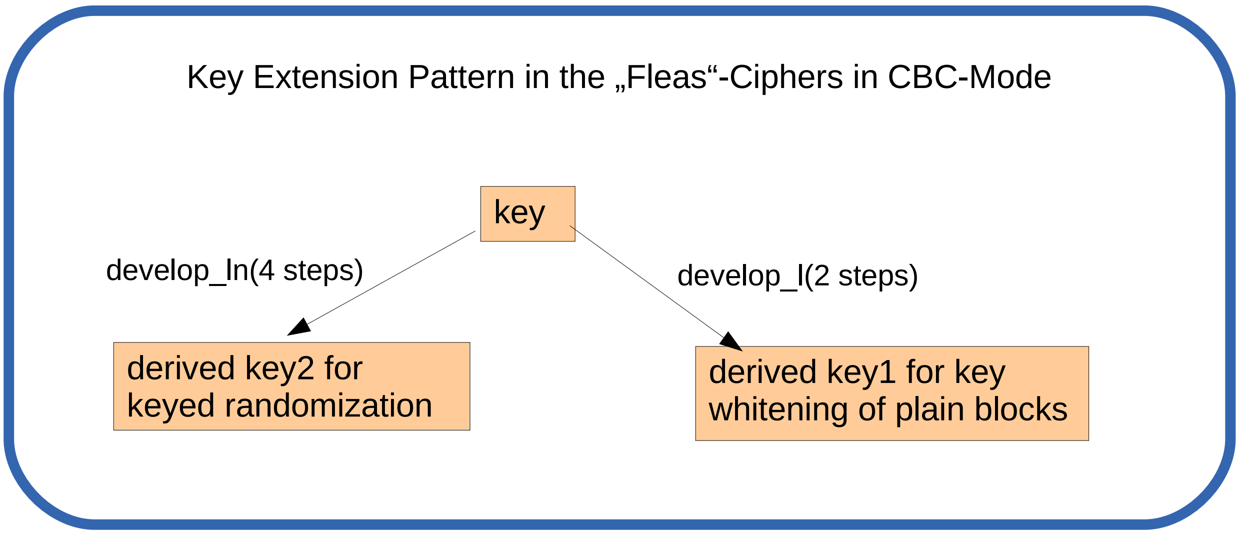
Enciphering of the blocks

Overview:

In any block except the first one, the block is xored with the last cipher block(CBC mode).

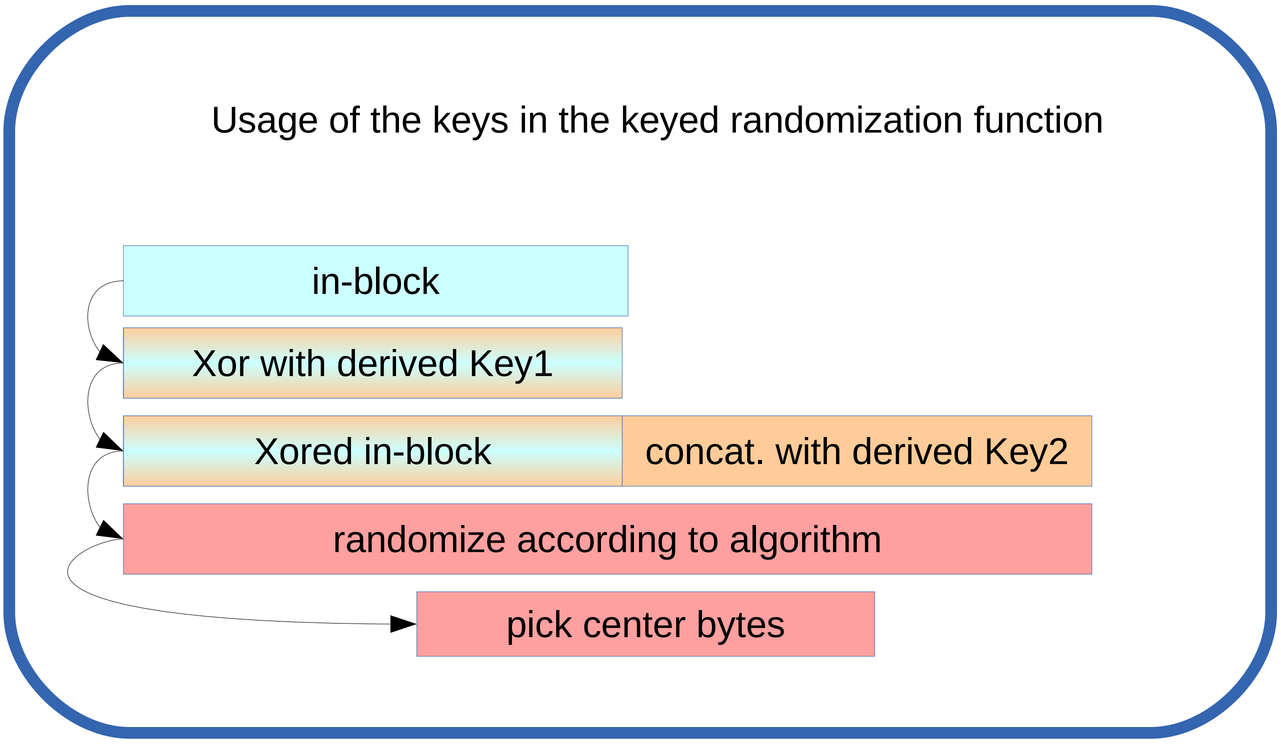
A derivative(not in the mathematical sense) of the key is prepared to be xored to the block later in the round function of the Feistel Network. If the block is larger than the key, key derivative bytes will be cyclically reused to cover the complete block(key whitening).

Another derivative of the key is calculated and stored for use as key in the Feistel Round Function.


Key Extension Pattern in the „Fleas“-Ciphers in CBC-Mode
key
develop_In(4 steps)
develop_I(2 steps)
derived key2 for keyed randomization
derived key1 for key whitening of plain blocks

The block then undergoes 4 rounds in a symmetrical Feistel Network. The round function consists of applying one of the Fleas pseudorandom functions to the concatenation of the second key derivative and the block. Block length number of bytes bytes of the center section of the pseudorandom functions output are selected as output of the round function.

This is done to all blocks.


Usage of the keys in the keyed randomization function
in-block
Xor with derived Key1
Xored in-block
concat. with derived Key2
randomize according to algorithm
pick center bytes

The routine "encipher_file" is located in the module "helpersxx.cpp". The code is not compact, somewhat involved and may be tedious to read. I apologize for that.

The procedure "decipher file" is located in the same module. It is not shown here since it basically performs the same operations as "encipher_file", just with negative parameter "feistelsteps" in procedure"dofeistel_n". The code of procedure "encipher file"is shown below:

```
/*******************************************************/
bool encipher_file(char* fnam, char* key, int keylen, char feistelsteps, int k_steps, int blolen,
int mode, wxString algo, unsigned char shoprog)
// verschlüsselt ein File mit key
// wahre Blocklänge 2*blolen, blolen muss größer 25 sein
// alte filelänge wird vorangestellt
{
  long long filen=0, blockzahl, j,newfilen,blinds,bytcount=0;
  FILE *plainfile, *cipherfile;
  char *bff1, *bff_o, *bff_last, *cpa_bl, *key2, *sfnam;
  int restlen,i,cpa_blen, algoflag;
  int currpos, lastmax=0;
  bool goodp=true;
  aes_context aesc;
  unsigned int algocode;
  wxString fnwx;


  if( (algo==_("F_cnt_1c"))||(algo==_("F_cnt_1d"))||(algo==_("F_cnt_1b")) ) //-> countermode
  {
      return( encipher_file_cntmode_mt(fnam, key, keylen, k_steps, blolen, 2, algo, shoprog) );
  }
  algocode=get_code_4_algo(algo); //
  if((blolen<26)&&(algo!=_("aes"))) return false;// kein Platz für die Filelänge!
  if(strlen(fnam) > 398) return false;
  fnwx = wxString::FromAscii(fnam);
  if(!wxFile::Exists(fnwx))
  {
      throwout(_("File does not exist!!\nAborting."),2);
      return false;
  }
  //determine Length
  wxFile tmpfl(fnwx);
  filen=tmpfl.Length();
  tmpfl.Close();
  //now normal old code
  if((plainfile = fopen(fnam, "rb"))== NULL) return false;
  // jetzt cipherfile öffnen mit anhängsel "_c2" etc und setze algoflag
    //create new sfnam "shadow fnam" buffer
    sfnam=(char *)malloc(strlen(fnam)+10);
    strcpy(sfnam,fnam);
  if(algo ==_("Fleas_1_8"))   {strncat(sfnam,"_c3",3); algoflag=0;}
    else if(algo ==_("Fleas_3"))   {strncat(sfnam,"_f3",3); algoflag=3;}
     else if(algo ==_("Fleas_4"))   {strncat(sfnam,"_f4",3); algoflag=4;}
     else if(algo ==_("Fleas_5"))   {strncat(sfnam,"_f5",3); algoflag=5;}
     else if(algo ==_("Fleas_x2"))   {strncat(sfnam,"_x2",3); algoflag=6;}
     else if(algo ==_("Fleas_x5"))   {strncat(sfnam,"_x5",3); algoflag=7;}
     else if(algo ==_("Fleas_o2"))   {strncat(sfnam,"_o2",3); algoflag=8;}
     else if(algo ==_("Fleas_o5"))   {strncat(sfnam,"_o5",3); algoflag=9;}
     else if(algo ==_("Fleas_l"))   {strncat(sfnam,"_l",2); algoflag=10;}
     else if(algo ==_("Fleas_ls"))   {strncat(sfnam,"_ls",3);  algoflag=11;}
     else if(algo ==_("Fleas_l3"))   {strncat(sfnam,"_l3",3);  algoflag=13;}
     else if(algo ==_("Fleas_lc"))   {strncat(sfnam,"_lc",3);  algoflag=14;}
     else if(algo ==_("Fleas_ld"))   {strncat(sfnam,"_ld",3);  algoflag=15;}
     else if(algo ==_("Fleas_lb"))   {strncat(sfnam,"_lb",3);  algoflag=16;}
     else if(algo ==_("aes"))   {strncat(sfnam,"_ae",3);blolen=8;algoflag=12;}
       else{throwout(_("unknown algorithm! \n aborting (1)"));free(sfnam);return false;}
  if((cipherfile = fopen(sfnam, "wb"))== NULL)
  {
    free(sfnam);
    return false;
  }
  free(sfnam);

//initialize Progress bar
  wxProgressDialog pbar(_("Enciphering Progress"),_("Computer is working hard :-)"),100);

  cpa_blen=blolen; //cpa_blen is matching blolen
  /** blolen must be 8 for aes!!  **/
  if((algo == _("aes"))&&(blolen != 8))
  {
      throwout(_("fatal error in aes blocksize!")); return false;
  }
  //create cpa_blocker block and possibly derived key
  key2= (char *) malloc(keylen);
  memcpy(key2,key,keylen); //key2 filled for use
  switch( algoflag)
  {
    case 4 :
      // build derived key for use with Fleas_4
      develop_c(key2,keylen,2);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_b_malloc(keylen, key,4, cpa_blen);
      break;

    case 5 :
      // build derived key for use with Fleas_4
      develop_x(key2,keylen,2,3);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_b_malloc(keylen, key,4, cpa_blen);
      break;

    case 6 :
      // build derived key for use with Fleas_x2
      develop_x1(key2,keylen,2,3);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_x_malloc(keylen, key,4,2, cpa_blen);
      break;

    case 7 :
      // build derived key for use with Fleas_x5
      develop_x1(key2,keylen,2,3);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_x_malloc(keylen, key,4,5, cpa_blen);
      break;

    case 8 :
      // build derived key for use with Fleas_o2
      develop_o(key2,keylen,2,3,0);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_o_malloc(keylen, key,3,2, cpa_blen,0);
      break;

    case 9 :
      // build derived key for use with Fleas_o5
      develop_o(key2,keylen,2,3,2);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_o_malloc(keylen, key,3,5, cpa_blen,2);
      break;

    case 10 :
      // build derived key for use with Fleas_l
      develop_l(key2,keylen,2);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 3,cpa_blen);
      break;

    case 11 :
      // build derived key for use with Fleas_ls
      develop_l(key2,keylen,2);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 3,cpa_blen);
      break;

    case 12 :
      // build derived key for use with aes
      develop_o(key2,keylen,2,3,2); //not used
      // build cpa_blocker with different algorithm
      cpa_bl= develop_o_malloc(keylen, key, 3,5,6*blolen,2); //cpablen, erste 16 für cpablock, 2.&3.
16 für key in 2 rounds
      break;

    case 13 :
      // build derived key for use with Fleas_l3
      develop_ls(key2,keylen,3,3,1);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 3,cpa_blen);
      break;

    case 14 :
      // build derived key for use with Fleas_lc
      develop_ln(key2,keylen,3,3,1);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 3,cpa_blen);
      break;

    case 15 :
      // build derived key for use with Fleas_ld
      develop_ln2_mt(key2,keylen,3,3,1);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 4,cpa_blen);
      break;

    case 16 :
      // build derived key for use with Fleas_ld
      develop_ln2_mt(key2,keylen,3,3,1);
      // build cpa_blocker with different algorithm
      cpa_bl= develop_l_malloc(keylen, key, 2,cpa_blen);
      break;

    default:
      cpa_bl=bytmixxmalloc(keylen,key,6,cpa_blen);
      break;
  }


//neue Routine zur Filelaenge already set filen
  /*while (!feof(plainfile)) {   // bestimme die Filelänge
      fgetc (plainfile);
      filen++;
      }
      filen--; //eins zuviel gezaehlt!
      rewind(plainfile);
*/
  //bestimme jetzt die Blockzahl
  blockzahl = (filen + 50) / (2*(long long)blolen);
  //
  if( ((filen + 50) % (2*(long long)blolen)) !=0) { blockzahl++; } //jetzt ham wir die richtige
Blockzahl
  newfilen = blockzahl*2*(long long)blolen; //neue Filelaenge
  blinds = newfilen - filen;
  bff1 = (char *)malloc(2*blolen); bff_o = (char *)malloc(2*blolen);  //Buffer in and out allozieren
  if(mode == 1) bff_last = (char *)malloc(2*blolen); //bei CBC-mode noch ein zusätzlicher Buffer
  //srand((unsigned int) *(bff_o)); // randomvariable mit unbestimmtem seeden

#if defined(__WXMSW__)
  sprintf(bff1,"si3:%Ld ",filen); // hat jetzt die echtlänge an den Anfang geschrieben
#else
  sprintf(bff1,"si3:%lld ",filen); // hat jetzt die echtlänge an den Anfang geschrieben
#endif
  // jetzt restlänge bestimmen
  restlen=(int)(filen-(blockzahl-1)*2*(long long)blolen);
  //differenz mit Zufallszahlen auffüllen
  // kann auch negativ sein: dann Zufallszahlen AUCH in den nächsten Block eintragen
  bytcount=strlen(bff1);
  for( i=strlen(bff1);i<2*blolen;i++) //von ende der Längenangabe bis blockende
  {
      bytcount++;
      if(bytcount>blinds) *(bff1 + i)= (char) fgetc(plainfile);
      else *(bff1 + i)= zuf.get_rbyte(); // mit zufallszahlen auffüllen
  } // füllt den Rest mit dem Filerest auf, dazwischen wirds mit zufallszahlen aufgefüllt und wird
bei decipher auch weggeschmissen
  // jetzt rausschreiben in cipherfile
  /********* muss noch verschl. werden ************/
//ersten block verschlüsseln
  if(algo==_("aes"))
  {
      //set encipher structure
        if (aes_setkey_enc( &aesc, (const unsigned char *)(cpa_bl+16), 256 )!= 0)
          throwout(_("error in setting aes key context"));
      //xor cpa_blocker to input block
      if(mode==1) for(i=0;i<16;i++) *(bff1+i) ^=  *(cpa_bl +i);
      //use aes in ecb mode
        if(aes_crypt_ecb( &aesc, AES_ENCRYPT,(const unsigned char *)bff1,(unsigned char*)bff_o)!=0)
throwout(_("error in aes transformation"));
  }
  else if(!dofeistel_n(2*blolen, keylen,bff1, bff_o, key2, feistelsteps,k_steps,cpa_blen, cpa_bl,
algocode))
      { printf("Error in dofeistel");}
  //else{throwout(_("fatal error in encipher file-> must debug properly !!!")); }


   //falls CBC-Mode in bff_last aufheben
  if(mode==1) memcpy(bff_last,bff_o,2*blolen);

  for( i=0;i<2*blolen;i++)
  {
     fputc((char) *(bff_o + i),cipherfile) ;
  } // schreibt das Produkt der ersten Blockverschl. in den ersten Block des Cipherfiles

  //jetzt das restliche file
  for(j=1;j<blockzahl;j++)
  {
       if(shoprog>0)
       {
         //prog bar propagation in main loop
         currpos=(int)( (100*j)/(blockzahl-1));
         if((currpos >lastmax)&&goodp)
         {
            goodp=pbar.Update(currpos);
            lastmax=currpos;
         }
       }

      for(i=0;i<2*blolen;i++)
      {
          bytcount++;
          if(bytcount <= blinds)
          {
              //*(bff1 + i)=(char) (rand() % 256);
              *(bff1 + i)=zuf.get_rbyte();
          }
          else
            *(bff1 + i)= (char) fgetc(plainfile);
      }
      // jetzt bei CBC mode XOREN mit vorigem Cipherblock
      if(mode == 1) //wenn in CBC Mode
      {
        for(i=0;i<2*blolen;i++)
        {
          *(bff1 + i) ^= *(bff_last + i);
        }
      }
      if(algo==_("aes"))
      {
        //use aes in ecb mode, CBC pattern already done
        if(aes_crypt_ecb( &aesc, AES_ENCRYPT,(const unsigned char*)bff1,(unsigned char*)bff_o)!=0)
throwout(_("error in aes transformation"));
       }
       else if(!dofeistel_n(2*blolen, keylen,bff1, bff_o, key2, feistelsteps,k_steps,cpa_blen,
cpa_bl,algocode))
          { printf("Error in dofeistel");}
// cipher in last sichern
      if(mode == 1) memcpy(bff_last,bff_o,2*blolen);
      for(i=0;i<2*blolen;i++)
      {
         fputc((char) *(bff_o + i),cipherfile) ;
      }
  }
  free(bff1); free(bff_o); free(cpa_bl), free(key2);
  if(mode == 1) free(bff_last);
  fclose(plainfile);
  fclose(cipherfile);
  return true;
}
/*****************************************/
```

The produre "dofeistel_n" called for each block does just what it says and will not be shown here. The round function of the Feistel network is called "roundfun". It is contained in the module helpersxx.cpp and calls a keyed randomization function, "k_develop_f4" for the more recent algorithms. This keyed randomization function is shown below:

```
/**********************************************/
bool k_develop_f4(int blolen, char* bladwell, int cpa_blen, char* cpa_bl, int keylen, char* keytxt,
int rounds, int algonum , memblo *mb)
// blolen lange bloecke für bladwell, cpa_blen fuer cpa_bl
//hier wird der keyblock auch im Algorithmus verwendet
//cpa_blocker and (derived?) key were precalculated
{
   unsigned int totlen;
   char *devbuff;
   int i;
   bool okflag=true;

   //check validity of algonum
   if(algonum>11 || algonum<0){
     throwout(_("unknown algorithm in k_develop_f4")); return false;}

   totlen= blolen+keylen; //total blocklength
   devbuff= (char *) malloc(totlen); //allocate working block
   //set first part of working block from xored bladwell and cpa_bl
   if((cpa_blen>0)&&(cpa_bl != NULL))
   {
      for(i=0;i<blolen;i++)
         *(devbuff+i)= *(bladwell+i) ^ *(cpa_bl + (i%cpa_blen));
   }
   else
   {
      memcpy(devbuff,bladwell,blolen);
   }
   //fill (possibly derived) key into rest of working block
   memcpy(devbuff+blolen,keytxt,keylen);
   //oneway-develop devbuff for rounds times
   switch( algonum)
   {
     case 0: //for Fleas_ 4
       if(!develop_e(devbuff, blolen, rounds)) okflag = false;
       break;

     case 1:   //for Fleas_5
       if(!develop_x(devbuff, blolen, rounds, 5)) okflag = false;
       break;

     case 2:  //for x2
       if(!develop_x1(devbuff, blolen, rounds,2)) okflag = false;
       break;
```

```
    case 3:  //for x5
      if(!develop_x1(devbuff, blolen, rounds, 5)) okflag = false;
      break;

    case 4: //for o2
      if(ispowof2(totlen)) //wenn totlen eine zweierpotenz
      {
          if(!develop_o(devbuff, totlen, rounds,2,0)) okflag = false;
      }
      else if(ispowof2(blolen)&&blolen>keylen)
      {
        if(!develop_o(devbuff, blolen, rounds,2,0)) okflag = false;
        if(!develop_o(devbuff+keylen, blolen, rounds,2,0)) okflag = false;
      }
      else
      {
        if(!develop_o(devbuff, totlen, rounds,2,0)) okflag = false;
      }
      break;

    case 5: //for o5
      if(ispowof2(totlen)) //wenn totlen eine zweierpotenz
      {
          if(!develop_o(devbuff, totlen, rounds,5,2)) okflag = false;
      }
      else if(ispowof2(blolen)&&blolen>keylen)
      {
        if(!develop_o(devbuff, blolen, rounds,5,2)) okflag = false;
        if(!develop_o(devbuff+keylen, blolen, rounds,5,2)) okflag = false;
      }
      else
      {
        if(!develop_o(devbuff, totlen, rounds,5,2)) okflag = false;
      }
      break;

    case 6:  //for develop_l
      if(ispowof2(totlen)) //wenn totlen eine zweierpotenz
      {
          if(!develop_l(devbuff, totlen, rounds, mb)) okflag = false;
      }
      else if(ispowof2(blolen)&&blolen>keylen)
      {
        if(!develop_l(devbuff, blolen, rounds,mb)) okflag = false;
        if(!develop_l(devbuff+keylen, blolen, rounds,mb)) okflag = false;
      }
      else
      {
        if(!develop_l(devbuff, totlen, rounds,mb)) okflag = false;
      }
      break;

    case 7:  //für Fleas_ls
          if(!develop_ls(devbuff, totlen, rounds,2,1)) okflag = false;
      break;

    case 8:  //für Hash: Fleas_lx3 und cipher l3
          if(!develop_ls(devbuff, totlen, rounds,3,1)) okflag = false;
      break;

    case 9:  //für Hash und cipher: Fleas_lc
          if(!develop_ln(devbuff, totlen, rounds,3,1)) okflag = false;
    break;

    case 10:  //für Hash und cipher: Fleas_ld jetzt multicore, 4 paths
          if(!develop_ln2_mt(devbuff, totlen, rounds,4,1)) okflag = false;
    break;

    case 11:  //für Hash und cipher: Fleas_ld jetzt multicore, 4 paths
          if(!develop_ln2_mt(devbuff, totlen, rounds,2,1)) okflag = false;
    break;

    default:
       throwout(_("unknown algorithm in k_develop_f4"));
       okflag=false;
  }
  if(!okflag) throwout(_("error in applying oneway-fn in k_develop_f4"));
  // overwrite bladwell with result from devbuff-center
  memcpy(bladwell,devbuff+keylen/2,blolen);
  free(devbuff); // free working block
```

```
   return okflag;
}
/**********************************************/
```

The reader may easily see, that the key is concatenated to the block which had been xored with the other derived key whitening block before. Then the concatenation is developed by one of the corresponding pseudorandom functions specified by the parameter "algonum". Finally block size bytes are taken from the center of the randomized concatenation and are used to replace in-block.

The newer algorithms flight, flightx and chimera are exclusively offered in CTR-mode.

c) Using Academic Signature's Ciphers in Counter Mode

Usage in Counter Mode is substantially simpler than in CBC-mode. Padding to a multiple of block size, key whitening and bookkeeping of original file size is not necessary. Thus there is no need to derive two separate keys from the primary key. All that is needed is an IV I.e. the initial counter value of size "block size" bytes. This IV does not even need to be kept secret. In Academic Signature it is enciphered anyways because the computational cost is negligible and it adds another obstacle to the attacker. The IV is enciphered by a symmetrical Feistel Network.

Enciphering can be realized in the procedure "encipher_file_cntmode". There is also a multi threaded version "encipher_file_cntmode_mt", which is used for the algorithms employing a more paranoid security overhead at the price of decreased speed. In order to increase efficiency I also created and now use more involved one pass versions for ecc use and advanced symmetrical enciphering, that interweave header creation or header processing and are considerably harder to read. They are compatible though and produce identical cipher files.

The single thread procedure "encipher_file_cntmode" as contained in Academic Signature ver b52 is printed below:

```
/******************************************************/
bool encipher_file_cntmode(char* fnam, char* key, int keylen, int k_steps, int blolen, int mode, 
wxString algo, unsigned char shoprog)
//k_steps: number of owf-steps,   mode:0-clear number, 1number of lonulock steps for number
//
{
  long long filen=0, blockzahl, j;
  FILE *plainfile, *cipherfile;
  char *sfnam, *iv, *ic,*ip,hc;
  int i,algoflag,ichar, currpos,lastmax=0;
  longnumber ivno;
  bool goodp=true;

//check new mt version now
 if( (algo== _("F_cnt_1c"))|| (algo== _("F_cnt_1d"))) return encipher_file_cntmode_mt( fnam, key, 
keylen, k_steps, blolen, mode, algo, shoprog);
 if( algo == _("threefish")) return encipher_file_threefish( fnam, key, keylen, k_steps, blolen, 
mode, algo, shoprog);  // R1)
 if(algo == _("chimera")) return encipher_file_fleafish( fnam, key, keylen, k_steps, blolen, mode, 
algo, shoprog);  // R2)
    //some primitive error checking
  if(strlen(fnam) > 398) return false;
  wxString fnwx(fnam, wxConvFile);
  if(!wxFile::Exists(fnwx))
  {
      throwout(_("File does not exist!!\nAborting."),2);
      return false;
  }
  //determine Length
  wxFile tmpfl(fnwx);
  filen=tmpfl.Length();
  tmpfl.Close();
  if((plainfile = fopen(fnam, "rb"))== NULL) return false;
  // jetzt cipherfile öffnen mit anhängsel "_1c" etc und setze algoflag
```

```
    //create new sfnam "shadow fnam" buffer
    sfnam=(char *)malloc(strlen(fnam)+10);
    strcpy(sfnam,fnam);
  if(algo ==_("F_cnt_1c"))   {strncat(sfnam,"_1c",3); algoflag=9;}
  else if(algo ==_("F_cnt_1d"))   {strncat(sfnam,"_1d",3); algoflag=10;}
  else if(algo ==_("F_cnt_1b"))   {strncat(sfnam,"_1b",3); algoflag=11;}
   else if(algo ==_("flight"))   {strncat(sfnam,"_fl",3); algoflag=15;}
    else if(algo ==_("flightx"))   {strncat(sfnam,"_fx",3); algoflag=16;}
     /*else if(algo ==_("aes"))   {strncat(sfnam,"_ae",3);blolen=8;algoflag=12;}*/
      else{throwout(_("unknown algorithm! \n aborting (0)"));free(sfnam);return false;}
  if((cipherfile = fopen(sfnam, "wb"))== NULL)
  {
    free(sfnam);
    return false;
  }
  free(sfnam);

//initialize Progress bar
  wxProgressDialog pbar(_("Enciphering Progress"),_("Computer is working hard :-)"),100);
  //determine blockzahl
  blockzahl=filen/blolen;
  //create IV from random then lock with key
  iv=(char *) malloc(blolen);
  ic=(char *) malloc(blolen);
  ip=(char *) malloc(blolen);
  zuf.push_bytes(blolen, (unsigned char *)iv);  //A)
  wxString primalgo; // the name of the cipher for the IV
  if(algoflag==9) primalgo=_("Fleas_lc");
  else if(algoflag==10) primalgo=_("Fleas_ld");
  else if(algoflag==11) primalgo=_("Fleas_lb");
  else if(algoflag==15) primalgo=_("flight");
  else if(algoflag==16) primalgo=_("flightx");
  else if(algoflag==17) primalgo=_("flightx");

  else throwout(_("Error! \nunknown primary algo in countermode enciphering"));


  if(mode>0)
  {
      dofeistel(blolen, keylen, iv,ic, key, 4, mode, 0, NULL, primalgo);//4Steps, mode-fold owf, no
cpa blocker
  }
  else memcpy(ic,iv,blolen);
  //write locked IV  B)
  for(i=0;i<blolen;i++)
  {
      fputc( *(ic + i),cipherfile) ;
  }
  //put IV into longnumber
  ivno.resizelonu(blolen+2);  // C)
  memcpy(ivno.ad, iv, blolen);
  ivno.setsize(true);
  //loop complete blocks with progbar update
  for(j=0;j<blockzahl;j++)  // D)
  {
       if((shoprog>0)&&(blockzahl>1))
       {
         //prog bar propagation in main loop
         currpos=(int)( (100*j)/(blockzahl-1));
         if((currpos >lastmax)&&goodp)
         {
            goodp=pbar.Update(currpos);
            lastmax=currpos;
         }
       }
      //read in plaintext block
      for(i=0;i<blolen;i++)
      {
          *(ip + i)= (char) fgetc(plainfile);
      }
      //get key-developed number block from ivno
      memcpy(ic,(char *)(ivno.ad),blolen);
      k_develop_f4(blolen, ic,0,NULL,keylen,key,k_steps,algoflag); //9 ->Fleas_lc 10 ->fleas ld
      //xor with plaintext block
      //write to cipher file
      for(i=0;i<blolen;i++)
      {

          hc=*(ic + i);
          hc ^= *(ip+i);
```

```
            fputc( hc,cipherfile) ;
        }
        //increment ivno
        ivno.inc();
    }  // E)
    //get rest block from plainfile
    //get key-developed number block from ivno
    //xor bytewise and write to cipherfile
    memcpy(ic,(char *)(ivno.ad),blolen);  // F)
    k_develop_f4(blolen, ic,0,NULL,keylen,key,k_steps,algoflag); //9 ->Fleas_lc 10 ->fleas ld
    i=0;
    do
    {
        ichar=fgetc(plainfile);
        if(ichar==EOF) break;
        hc=(char) ichar;
        hc ^= *(ic + i);
        fputc( hc,cipherfile) ;
        i++;
    } while(ichar != EOF);
    //close files
    fclose(cipherfile);  // G)
    fclose(plainfile);
      //free buffers
      free(iv); free(ip); free(ic);
    return true;
}
/*****************************************************/
```

Description:

A) After some initial sanity checks, memory allocation and progress bar initialization, the IV is filled with random bytes.

B) An encrypted version of the IV is created and written to the cipher file.

C) A longnumber, the counter, initially containing the IV is created(low byte at low address).

D) A for loop running through all complete blocks of the file is entered.

E) Continue with loop.

F) Then work on the last, probably incomplete block and xor plain text byte wise with the corresponding counter cipher byte until EOF is encountered.

G) Close cipher file and plain file. Free the buffers.

Remarks:

R1) As you can see at the procedure entry, threefish and the chimera are redirected to special routines. This is necessary since threefish, which is also one component of the chimera, requires additional structures not needed in my fleas and flight algorithms.

R2) As you can also see, the chimera is internally called "fleafish". I decided to use the name "chimera" for the outside world to respect the term "fish" as brand of the Schneir group and to avoid encroaching on the brandname.

d) Protecting the Integrity of the Symmetric Cipher

For two applications of the Fleas cipher, I see a necessity to protect the integrity of the cipher. This is in direct use of the symmetric ciphers. These might be vulnerable in transit to bit flipping attacks, if the attacker already knows part or all of the plain text. The other case is secured storage of the public keys on hard disk.

For these cases I developed a new enciphering procedure called "eax_enc_file". It is located in the module "helpersxx.cpp". Despite its name it does not adhere to the EAX standard but employs an hmac pattern (encipher then authenticate). I beg your pardon for the misleading naming. The hmac is calculated using the procedure "get_lonu_hmac_from_filename_x" which is also contained in "helpersxx.cpp".

In brief:

The key is co-developed with a nonce to a unique key derivative.

An hmac of the cipher file is calculated using this derived key.

In the direct symmetric cipher of Academic Signature, a header file is prepended to the cipher file containing a plain text reference to hmac algorithm name, hmac value and nonce value. Below you see an example header.

```
salt: 42b1e07f6fcda7c604a0f9cc93f2e88aeec66db18979f50164673dd744ecf1a0
algo: F_cnt_1b
stretch: 1 stretchdom_253
MAC: Fleas_lb
e309f1a4b395b8919c8ed1df7055714e02afcbd8718b1726c6b43e334168f8935891d5d3c25e31f80b64551fb789f0eb
nonce:
2d0ffb10f4919354007e58cfb2e365278dcd42358540e75c9549246f3ac45ee205569ea5f74336ed7671ebaa01cb2ada
end_header_info::
```

The identifiers "salt: ", "algo: " and "stretch: " refer to salt, symmetric algorithm and stretch factor and stretch domain, respectively. They do not relate to cipher integrity.

The identifier "MAC: " refers to the name of the hmac-algorithm followed by the hex-representation of the hmac value.

"nonce: " is followed by the hex-representation of the nonce.

"end_header_info::" is the tail marker of the header. The cipher text begins after this marker.